JANUARY 1985

PRICE $5.00

/c

THE JOURNAL FOR C USERS

VOLUME 2 • NUMBER 1

C standards committee

The American National Standards Institute's X3-J11 committee, the C standards committee, has been working since 1983 on an informal draft of a standard. Although the draft is still fluid (subject to change), this column will cover the significant events of the December meeting in Phoenix, Arizona.

The major decision made in the December quarterly meeting was not to release the draft for unofficial public comment. The prevailing sentiment in the committee was that insufficient time was available at the week-long meeting to deliberate some significant issues. Several causes contributed to the defeat of the motion to send the draft to public comment. For the first time, however, a consolidated draft of the standard was presented to committee members at the December meeting.

Early in the formation of the committee, the group identified and addressed three areas of the C language: a language definition section, an environment (hardware and operating system) section, and a standard library section. Subcommittees were formed, and each one drew a draft for its subject. In December the three separate drafts were drawn into a master draft and presented.

On consolidation, some inconsistencies and problems were noted. After considerable debate, the committee decided that the draft was not ready for public comment. The chairman in each section is reshaping that section's contribution for reincorporation. The reincorporated draft of the C language will then be mailed in January to the committee members. At that time, a paper ballot to release the draft for public comment will be cast. If passed, the draft will be available for public review and comment by March, 1985.

The second most significant action of the week was the removal of ranged **case** statements from the draft. Members agreed that the **case** range operator, tentatively the double period (..), would cause too many compiler parsing problems.

If you have any questions or comments about the draft that is evolving toward a C standard, write to us at */c*. We will be happy to pass responses and questions to the committee and answer questions in this publication.

In this issue

- **C standards comittee** 1
- **News** 1
- **Letters to the editor** 2
- **The getopt ( ) function** 4
- **Register variables** 9
- **January's C Quiz** 16
- **Libraries** 16
- **Documentation** 16

News

Compilers

A port of Aylcon Corporation's C68 compiler for the 68000 CPU has been announced by Eyring Research Institute for Eyring's Pdos operating system. The compiler provides direct access to more than 100 Pdos system calls.

According to the publisher, Pdos supports a subset of AT&T's UNIX I/O library to give users portability from UNIX, UNIX-like, and other operating systems using the C language.

The price of Pdos C is $750.

Eyring Research Institute
1455 W. Route 820 North
Provo, UT 84601
(801) 375-2434

Whitesmiths has announced that its C and Pascal compilers for the IBM 370 mainframes have been enhanced to provide optional cross-compiler support. The company also announced that the compilers are available under IBM's VM/CMS.

(Continued on page 16)

que™

Letters to the editor

Q. The asterisk character has me confused. I see it in several places but don't fully understand when I'm looking at the asterisk as a pointer or a character.

Jan Griffin
Los Angeles, CA

A. You have discovered one of the frustrations that many beginning C programmers face. The December, 1984, quiz was based on C's using identical characters several different ways. The asterisk is one of these characters.

When used in a declaration, definition, or cast, the asterisk says that the entity is a pointer. When used in a statement or operation, the asterisk tells the compiler to fetch what this entity points to, not the entity itself.

For example, of the four definitions

```
int i;          int *ip;
char c;         char *cp;
```

those in the right column are pointers. These two definitions define `ip` and `cp` to be pointers to an integer and character, respectively. The two definitions in the left column define `i` to be an integer and `c` to be a character.

In the declaration

```
extern char *p
```

the variable `p`, whose storage is allocated in a different file (which is what `extern` means) is declared to be a pointer to a character. In each of the three cases in which the asterisk was used, a pointer was defined or declared.

/c: The Journal for C Users, is published monthly by

Que Corporation
7999 Knue Road
Indianapolis, Indiana 46250

Address all editorial correspondence to The Editor, */c*, at the address above. All requests for special permissions should be addressed to The Publisher, */c*, at the same address.

Domestic subscriptions: 12 issues, $60.00; outside U.S.A., $80.00. Single copy price, $5.00; outside U.S.A., $7.00. All funds are U.S.A. dollars. Send subscriptions, change of address, fulfillment questions, or bulk orders to

Subscriptions
/c
P. O. Box 50507
Indianapolis, Indiana 46250

Copyright © 1985, Que Corporation. All rights reserved. No part of this periodical may be used or reproduced in any fashion whatsoever, except in the case of brief quotations embodied in critical articles and reviews, without the prior written consent of Que Corporation.

Editor-in-Chief: Chris DeVoney
Managing Editor: Paul Mangin
Editorial Director: David F. Noble, Ph.D.
Copy Editor: Virginia Noble
Circulation: Sharon Sears
Cover Design: Don Tiemeier of Listenberger Design Associates

/c and Que are trademarks of Que™ Corporation. IBM is a registered trademark of International Business Machines Corporation. MS-DOS is a trademark of Microsoft™, Inc. UNIX and AT&T are registered trademarks of American Telephone and Telegraph Corporation.

In the next example, the asterisk is used three times. The first use will differ from the latter two.

```
char buf[] + "abcdefg";

main()
{

     char *p
     ...

     p = buf;

     while(*p++)
          putchar(*p);
```

The first time the asterisk is used, we are making a definition. `p` is defined as a pointer to a character.

The second and third uses of the asterisk is *indirection*; `*p` fetches what `p` points to, not the value of `p` itself. The second and third uses show `p` as a pointer.

Why wasn't an asterisk used in the following line?

```
p = buf;
```

The reason is that we wanted to place the address of the character array `buf` into `p`. When assigning a value to `p`, we want to change the value of `p`, not the entity that `p` points to. As we must change `p` itself, we do not use indirection, and the asterisk *cannot* be used.

If we say

```
p1 = p;
```

we have assigned the value in `p` (the address of `buf`) to `p1`. In this case, `p1` should be defined as `char *p1`

The line

```
p1 = *p;
```

assigns to `p1` the character pointed to by `p`. If this assignment is what we want, `p1` should be declared as `char p1` (or `int p1`).

The line

```
*p1 = *p
```

assigns the character pointed to by `p` to the location (character) pointed to by `p1`. Here `p1` must also be declared as `char *`.

When used in casting, the asterisk has the same meaning as in a declaration.

In the fragment

```
char *p = 0x80;
int *i;

i = (int *)p;
```

the first two lines define pointers. `p` is also assigned to point to memory location 80 hexadecimal. The third line makes `i` point to the same location as `p`. The

(Continued on page 3)

asterisk inside the cast [(int *)] informs the compiler that the value of p should be shaped into a pointer to integer before assigning this value to i. In essence, making a cast is like making a declaration or definition. The difference is that a cast is an operation informing the compiler to change one data type into another.

In the fragment

```
char *p;
int i;
i = *(int *)p;
```

we are using the first asterisk in the third line for indirection, and the second asterisk in that line for the cast. The third line verbalizes as "Take the memory location pointed by p. Change this temporary value into a pointer to integer [(int *)]. Fetch the integer pointed to by the temporary pointer (the asterisk outside the cast) and assign it to the integer i."

Don't be overwhelmed by this information. The dual use of the asterisk is confusing and takes time to understand. Fortunately, you can ease yourself into this understanding.

Use of the asterisk has almost a simple rule. An asterisk in a declaration, definition, or cast means to make the entity a *pointer*. At any other place in a program, the asterisk means *indirection*, or to take the object pointed to by the entity, not the entity (pointer) itself.

Q. I see that some programs use `#include "filename"`, and others use `#include <filename>`. Is there a difference between using double quotation marks and the less-than, greater-than signs?

Carol Struling
Santa Cruz, CA

A. Maybe. The difference is found in the searching order for an included file, and the difference depends on your compiler.

The UNIX compiler uses `"filename"` to state that the compiler should first search the directory holding the original file being compiled, then search the sequence of other places for the file `filename`. `<filename>` searches only the standard places, not the directory of the original file being compiled.

Because the preprocessor is CPU- and operating system-specific, and compiler publishers have latitude in implementing this `#include` directive, the action of a particular compiler to "" or <> is currently nonstandard.

Some compilers treat the two forms the same. Other compilers will not accept the <>. Some compilers respect the UNIX meaning of the two and perform searches for the included file(s) based on the way UNIX searches: either the same directory as the source file, then standard places; or just standard places.

By the way, the concept of *standard places* does not have a universal meaning. The standard places for the compiler to search are implementation-defined (defined by the compiler publisher).

The general practice is to use `#include <filename>` when you are using any of the standard header files (.h files) provided with the compiler or system. Use `#include "filename"` when the included file is special or unique to this program.

The reason for this practice is obvious. You have redefined some values in `stdio.h` for a particular set of programs. This new `stdio.h` should be in the same directory as are the programs that will use this new version. All of these files should use `#include "stdio.h"` to include the special version of this header file.

If you have any other programs in the same directory with `#include "stdio.h`, these programs, when recompiled, will use the new version of the header file. The outcome of this arrangement may not be desirable. If the other files used `#include <stdio.h>`, the standard header file will be included instead.

Q. I am slightly confused by `extern`. When should I use `extern` in a data declaration?

Matt Bartos
Tempe, AZ

A. Use the keyword `extern` whenever you need to declare an entity whose storage is allocated somewhere else, usually but not always, in a different file. Do not use `extern` when you define an item.

A good example is the `getopt()` function presented in this issue. In the file for `getopt()`, two variables, `int optind` and `char *optarg`, are defined. The storage for `optind` and `optarg` is allocated in the `getopt()` file. When the `getopt.c` file is compiled, space will be allocated for the two variables in this module.

The test program `tstgo.c` declares what `optind` and `optarg` are (`int` and `char *`) but also declares that the variables are external. By declaring these two entities as `extern`, the compiler will not allocate space for them in this module.

When the program is linked, the linker will resolve the `extern` statements in `tstgo.c` with the variable definitions in `getopt.c` and produce a working program.

Note that if you declare an item to be `extern`, one of the files that make up your program *must* define (allocate) space for the item.

The getopt() function

getopt() is a UNIX library function that gets the option letter (*switches*) from command line arguments. The getopt() presented here may be added to your non-UNIX library.

This version of getopt() uses strchr(), which is a part of the UNIX standard library. A version of strchr() is also presented.

Description of getopt()

```
int getopt(argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind;
```

getopt() returns the argv's next option letter that matches a letter in optstring.

optstring is a string of recognized option letters. if a colon follows an option letter in optstring, the option is expected to have an argument. The option argument may or may not be separated from the option letter by white space. optarg is set to point to the start of the argument on return from getopt().

optind reflects the index of the next argv to be processed. As optind is external, it is initialized to Ø automatically.

After all options have been processed (that is, when the first nonoption argv is processed), an EOF is returned. The special option -- is used to signal the end of options. When -- is encountered, EOF is returned, and optind points to the next argv. An option letter must be placed between the program name and additional arguments on the command line, as required by normal UNIX style.

Diagnostics

If an option letter not specified in optstring is encountered, getopt() prints a message on stderr and returns a question mark (?).

The colon may not be an option character. If you use a colon as an option character on the command line, the colon will be handled as an unrecognized option letter. The dash is also illegal as an option letter, except when the dash appears as the option indicator (first character in an argv) or as the end-of-options indicator (--). The colon and dash may appear in an option argument.

If an option specifies an argument (that is, if a colon follows the option letter in optstring), and no option string is found from the command line, getopt() prints an error message and returns a question mark.

Description of strchr()

```
char *strchr(s , c)
char *s;
int c;
```

strchr() returns a pointer to the first occurrence of the character c in the string s. If c is not found, NULL is returned. The null character is expected to terminate the string s.

getopt() program flow

getopt() is a simple but complex function. It is simple because it pulls option letters from command line arguments (-a, -bcd, etc.). The function is complex in that it must keep track of which argv (and where in the argv) it will continue processing. The function's complexity persists because an option, not *optional*, argument may or may not fall in the same argv where the option character falls.

The function starts by including stdio. h. This will give the proper definition for EOF and for the error message I/O. Next, optarg and optind are defined. A third variable, index, is defined. This variable will keep track of the next character to process in the argv. The reason for defining the variable as static is that, along with making the variable global (outside a function), the definition is hidden from all other files. If this variable were only global, a possible conflict could arise with any other function that declared a global variable called index.

After the declaration of the function, four items are declared. i is an int that will hold the character from argv to be tested. optflag is a flag that will hold information indicating whether a valid option letter has been found. char *p is a pointer that will hold the result of strchr(). Because strchr() returns a result that is not an int, strchr() is declared.

The function starts by zeroing both optarg and optflag. This action makes optarg a null string and defaults optflag to mean that a valid option character has not been found.

A while loop is used to control the function. The reason for the while will be explained shortly.

i is assigned from argv[at the current optind][at the current index character].

Because a colon is an illegal option character, i is immediately tested for a colon. If a colon is found, the null statement [if(i == ':');] causes the function to skip the code in the else if(index) statement.

If i is not a colon, the else if(index) statement tests whether index, the current character count in the current argv, is nonzero. If index is nonzero, we are testing a second or later character in the argv. strchr() is used to test whether i occurs in optstring.

The error message printed to stderr uses argv[0] and the illegal (unrecognized) character or the character that required a missing option argument.

If the index is zero, we are testing the first character in a fresh argv. If the first character is not a dash, then argv is not an option argument. Because this argv will be the next to process, we immediately return with an EOF. The test for a second dash adjacent to the first one is for the end-of-options indicator (--). This action will allow getopt() to get optind and index to the next argv. If the second character is also a dash, i is set to EOF, and we fall through. If the second character is not a dash, we will loop back to the while to retest the next character by means of the continue statement.

As with all other parts of this function, optind and index are set to the next character to be processed. Note that index has been incremented within the test for the second dash and is at the proper value for continuation of the while loop.

At this point in the function, all tests for valid option characters have been performed.

The next test

```
if(optflag && *(p + 1) == ':')
```

performs two functions. If optflag is nonzero, a valid option character has been found [if(optflag)]. We still must test to see whether an option argument is expected. The test if((*p+1) == ':') checks whether the next character after the option letter is a colon. p retains the pointer to the matched option letter in optstring. *(p + 1) returns the next character in the option string. Both conditions of the if test must be true, or this section of code is skipped.

By definition, the option argument may be in the same argv or in the next argv. The code

```
if(argv[optind][++index])
```

checks whether the character following the option letter in the argv is non-null. If the character is non-null, optarg is set to point to the beginning of the option argument in the current argv. Otherwise, optarg is set to point to the next argv. In either case, optind is incremented to point to the argv following the option argument, and index is set back to the first character in that argv.

if(!optarg) tests for a null string pointer. This tests whether a valid option argument follows the option character in the same argv or in the next argv (in other words, if the next argv is a null pointer). The final pointer in the array of pointers that argv represents is always a null. A null always signals the end of the command line arguments. If optarg is null, no option argument was provided on the command line. In this case, an error message is printed, and i is set to a question mark.

The final code in the if(optind && *(p + 1) == ':') test returns i. Because optind and index have been properly set, we now safely return from the function.

If we have not returned from the function, the function increments index, setting it at the proper value for the next entry into getopt(). However, if the next character in this argv is a null, we must advance optind to point to the next argv, and index must be reset to point to the first character in the argv. This action must also be taken if we have a condition that made i an EOF. If i is EOF, we must set optind and index to point to the proper value for the next argv.

At this point, if we are still within the while loop, getopt() returns whatever is in i.

If for any reason we have not yet returned from the function, we will return an EOF. This point can only be reached if we encounter an argv with a single dash or if the next argv points to a null, the normal terminator for **argv.

Modifications and use

The most predominant modification to getopt() is deleting the error messages. If you do not wish to print an error message, simply delete the two fprintf() functions. Because this version of getopt() is different, you may be wise to store this function in a personal library or give the function a name other than getopt() in your standard library.

If your compiler does not set argv[0] to the program name, you can modify the calling program with the lines

```
char prog_name[] = {"tstgo"};      /* program name */

main(argc, argv)

{

  ...

  argv[0] = prog_name;             /* force argv[0] to
                                      program name */
```

This modification has been done to tstgo.c. The lines are currently commented out. If you wish to use these lines, remove the comments.

For CP/M and similar environments, use only capital letters for optstring. CP/M automatically changes letters on the command line to uppercase characters. Under CP/M and similar environments, getopt() will never match a command line option character that is written in lowercase in optstring.

If your o/s (operating system) passes the command line arguments without alteration, you might duplicate the option letters in optstring in both upper- and lowercase characters. If you enter only lowercase characters in optstring, then a correct command line option letter typed in uppercase will not be recognized. Instead of modifying getopt() to perform a tolower() or toupper(), change optstring.

A non-null pointer indicates a match, and `optflag` is set to nonzero to reflect that a match has been found. If no match is found (that is, if `strchr()` returns a null pointer), no action is taken, and the function falls through the remaining tests.

The tstgo.c test program

The program `tstgo.c` is a simple program that tests the result of `getopt()`. After compiling and linking the program with `getopt()` and `strchr()`, execute the program with one or more arguments. This test program will help you understand the results of `getopt()`.

Code fragment

The provided code fragment is a demonstration of a possible use of `getopt()`.

getopt.c

tabs = 3

```
/*                                                    */
/*    getopt() -  get option letter from argv         */
/*                                                    */
/*    environment independent                         */
/*                                                    */


#include "stdio.h"

char *optarg;              /* pointer to option argument */
int optind = 0;            /* count of next argv to process */
static int index;          /* private counter for next char in argv to process*/

int getopt(argc, argv, optstring)
int argc;
char *argv[];
char *optstring;
{

   int i;                        /* int to hold option letter     */
   int optflag;                  /* opt flag, 1 if good, 0 if bad */
   char *p;                      /* pointer for strchr()          */
   char *strchr();               /* char search function          */

   if(!optind)
      ++optind;
   while(argv[optind]) {                  /* while an argv exists */
      i = argv[optind][index];            /* get next char */
      if(i == ':');                       /* colon is wrong */
      else if(index) {
         if(p = strchr(optstring, i))     /* non-null means match */
            optflag = 1;
      } else {
         if(argv[optind][index] != '-') {       /* if first char not a '-' */
            return(EOF);                        /* return immediately */
         } else {
            if(argv[optind][++index] == '-') {  /* second char a '-' */
                  i = EOF;
            } else {
               if(!argv[optind][index]) {       /* next char a null   */
                  optind++;                     /* point to next argv */
                  index = 0;
               }
               continue;                        /* the loop */
            }
         }
      }
```

```
      if(optflag && *(p + 1) == ':') {          /* option argument ? */
         if(argv[optind][++index])              /* where the argument? */
            optarg = &argv[optind][index];
         else
            optarg = argv[++optind];
         optind++;                              /* set optind & index */
         index = 0;
         if(!optarg) {                          /* no argument */
            fprintf(stderr, "%s: option requires an argument == %c\n",
               argv[0], i);
            i = '?';
         }
         return(i);                             /* return now */
      }
      ++index;                                  /* bump index */
      if(i == EOF || !argv[optind][index]) {    /* EOF or null? */
         optind++;                              /* adjust optind & index */
         index = 0;
      }
      if(!optflag && i != EOF) {                /* no match */
         fprintf(stderr, "%s: illegal option -- %c\n", argv[0], i);
         i = '?';
      }
      return(i);                                /* if there was a char */
   }
   return(EOF);                                 /* if we run out of argvs */

}
```

strchr.c

tabs = 3

```
#include "stdio.h"

char *strchr(s, c)      /*return pointer to c in string s */
char *s;
int c;
{

   while(c != *s && *s)
      ++s;
   if(*s == NULL)
      return(NULL);
   else
      return(s);

}
```

tstgo.c

tabs = 3

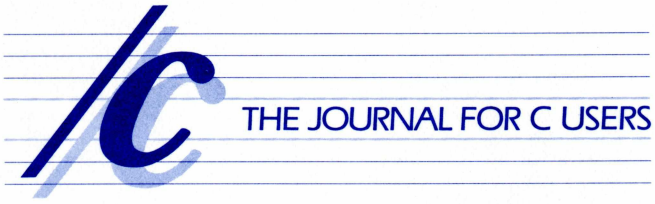
```
/*    tstgo.c  -- test getopt()*/

#include <stdio.h>

char *opts = {"AaBbCcD:d:EeF:f:"};

/* comment in the following line if your compiler does not set argv[0] */
/* to your program name */

/*char prog_name[] = {"tstgo"};*/
```

```
main(argc, argv)
int argc;
char **argv;
{

   extern int getopt();
   extern char *optarg;
   extern int optind;

   int i;

   /* comment in next line if you commented in prog_name */
   /* argv[0] = name; */

   while((i = getopt(argc, argv, opts)) != EOF) {
      printf("getopt() = %c, optarg = %s, optind = %d\n",
         i, optarg ? optarg : "zero" , optind);
   }

}
```

Code fragment showing use of getopt( )

```
. . .

   int pagesize, lines, c;
   char pathname[63];

   extern int *optind;
   extern char *optarg;

   while((c = getopt(argc, argv, "P:p:L:l:Ff") != EOF)
      switch(c)|{
         case 'p':
               pagesize = atoi(optarg);
               break;

         case 'P':
               strcpy(pathname, optarg);
               break;

         case 'L':
         case 'l':
               lines = atoi(optarg);
               break;

         case 'F':
         case 'f':
               formfeed = TRUE;
               break;

         case '?':
               fprintf(stderr, "usage .....");
               exit(2);


      }
   }
. . .
```

SUBSCRIBE TO . . .



You don't want to be without valuable, up-to-date information on the C language. Each monthly issue provides:

- Valuable C programming techniques
- Clear, concise explanations of functions
- Exciting tips on the C language
- And more . . .

A "moderately technical" journal, ***/C*** is sophisticated enough for the experienced C programmers, yet within reach of newcomers.

This convenient, postage-free card makes subscribing easy. Just fill in the order form below and mail it with your payment. Or check the box marked "Bill Me," and your payment won't be due until after you receive your first issue. For faster service, call our toll-free order line (**1-800-428-5331**) and use your credit card information TODAY. Take your pick of payment options, but subscribe NOW!

CALL NOW OR USE THIS CONVENIENT FORM TODAY.

(CUT HERE)

YES, I want */C.*

☐ Send me 12 monthly issues for $60 (outside U.S.A., $80)*

☐ Send me 24 monthly issues for $110 (outside U.S.A., $145)*

*All money must be payable in U.S. funds.

Company Name (if applicable) _____

Attention of: (Name) _____

Address _____

City _____ State _____ ZIP _____

Method of Payment:

☐ Check ☐ VISA ☐ MasterCard ☐ AMEX ☐ Bill Me

Cardholder Name _____

Card Number _____ Exp. Date _____

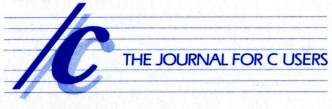
Signature _____

(for credit card)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First Call Permit No. 9918 Indianapolis, IN

Postage will be paid by addressee



P.O. Box 50507
Indianapolis, IN 46250

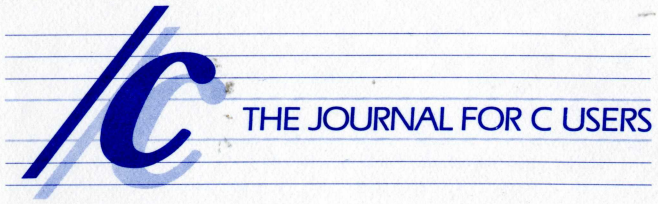
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First Call Permit No. 9918 Indianapolis, IN

Postage will be paid by addressee

P.O. Box 50507
Indianapolis, IN 46250

SUBSCRIBE TO . . .



You don't want to be without valuable, up-to-date information on the C language. Each monthly issue provides:

- Valuable C programming techniques
- Clear, concise explanations of functions
- Exciting tips on the C language
- And more . . .

A "moderately technical" journal, ***/C*** is sophisticated enough for the experienced C programmers, yet within reach of newcomers.

This convenient, postage-free card makes subscribing easy. Just fill in the order form below and mail it with your payment. Or check the box marked "Bill Me," and your payment won't be due until after you receive your first issue. For faster service, call our toll-free order line (**1-800-428-5331**) and use your credit card information TODAY. Take your pick of payment options, but subscribe NOW!

CALL NOW OR USE THIS CONVENIENT FORM TODAY.

(CUT HERE)

YES, I want */C.*

☐ Send me 12 monthly issues for $60 (outside U.S.A., $80)*

☐ Send me 24 monthly issues for $110 (outside U.S.A., $145)*

*All money must be payable in U.S. funds.

Company Name (if applicable) _____

Attention of: (Name) _____

Address _____

City _____ State _____ ZIP _____

Method of Payment:

☐ Check ☐ VISA ☐ MasterCard ☐ AMEX ☐ Bill Me

Cardholder Name _____

Card Number _____ Exp. Date _____

Signature _____

(for credit card)

Register variables

C allows the use of `register` variables. A `register` variable is an entity that will be heavily used. If possible, the compiler will place such variables in the registers of the computer's CPU. Programs can therefore be smaller in size and execute faster. The restrictions on register variables are threefold: C language-specific, CPU-specific, and implementation-specific.

C language restrictions

The C language places two restrictions on `register` variables. The first restriction is that only automatic variables or formal parameters of a function may be a `register` variable. In the example

```
int checkline(s, n)
register char *s;
int n;
{
     register int i;
     register char c;
     ...
}
```

`s`, `i`, and `c` are register variables. `s` is a parameter to a function, and `i` and `c` are automatic variables. If desired, `int n` can also be defined as `register`.

Note that in the function calling `checkline()`, `s` does not need to be defined as a `register` variable. The compiler will automatically move `s` into a CPU register when `checkline()` is invoked.

The second restriction is that you cannot take the address of a `register` variable. In other words, you cannot use a pointer to access a `register` variable. By inference, the inability to take the address of a register effectively eliminates using a register to hold a structure or union. Because both of these must be passed and accessed through a pointer, a register cannot hold a structure or union. Even UNIX System V compilers do not allow `register` structures although these compilers do allow passing of the actual structure or union on the stack.

The contents of an array are always manipulated through a pointer. For this reason, an array cannot be held in a `register` variable. However, a pointer to an array can be placed in a register. Pointers to structures or unions can also be placed in a `register` variable. The subtle differences among these will be discussed later in this article.

CPU restrictions

The number and uses of `register` variables are further restricted by the number of CPU registers available. A `register` variable needs a CPU register that is not undergoing dynamic use by the executable program. Therefore, the accumulator, flag, stack pointer, and program counter registers, as well as one or two additional CPU registers, are typically excluded.

On 8-bit CPUs one CPU register may be available for use with `register` variables. Sixteen-bit CPUs may offer one to four registers, and 32-bit CPUs may offer as many as six registers for use with `register` variables.

The key point to remember about CPU restrictions is that the number of potentially available registers for use with `register` variables will vary among differing CPUs.

Implementation restrictions

Each implementation can place further restrictions on `register` variables: whether `register` variables are recognized and allowed, how many variables can be placed in registers, what types of variables can be made `register`, and what occurs to excess or disallowed `register` definitions.

No implementation of C is bound, or obligated, to use `register` variables. A `register` is an optional data type that can be implemented by the compiler publisher, but the publisher is not obligated to offer a `register` service in the compiled programs.

The number of `register` variables offered can vary among implementations. In practice, most 8-bit compilers offer none, or they offer one or two `register` variables. Sixteen-bit compilers offer none or one to three `register` variables. Most 32-bit compilers can use four to six `register` variables at a time. Most compilers document in their manuals the maximum number of `register` variables that may be used.

The types of data that can be placed in registers are restricted by C. No array, structure, or union may be placed in a register. Implementations may place further restrictions on data types that can be held in registers. The common restrictions allow that only `int`s, `char`s, and pointers can be placed in registers. Some 16-bit implementations further restrict pointers to small model (16-bit) pointers, disallowing large model (32-bit) pointers.

The reaction of compilers to disallowed or excess `register` declarations is vaguely defined by K & R but is actually implementation-defined. K & R states that the word `register` is ignored for excess or disallowed declarations. How "the word `register` is ignored" is defined by the compiler publisher.

Compiler publishers that do not allow `register` variables usually drop the word `register` into the bit bucket. The word `register` simply disappears as the compiler translates the source code, and the `register` declaration is instead made an automatic declaration.

Compiler publishers that do recognize and handle `register` variables normally fulfill as many legal declarations as possible in the order in which the declarations are encountered. Excess declarations are converted to `auto`. To date, no compiler, when encountering excess declarations, will prioritize the declarations and fulfill the most meaningful `register` declarations—those that yield the best performance.

Disallowed or illegal declarations

A problem can exist with a `register` declaration that should be disallowed but is not. Consider the following small code fragment:

```
{
        int i;
        register struct count something;

        i = something.times;
        ...
}
```

The line `register struct count something` is in illegal definition. The result of this line, however, is implementation-specific and can widely (and wildly) vary. The answer hinges on the definition of "disallowed declarations."

Compilers that simply ignore the word `register` will convert `struct count something` into an automatic variable. The program will compile without a diagnostic error.

Compilers that respect the word `register` but define a *disallowed declaration* as an impossible declaration will ignore the word `register`. Such a compiler makes `something` an automatic variable. A warning may or may not be displayed.

Compilers that respect the word `register` but define a *disallowed declaration* as a register declaration that cannot be fulfilled will give an error message. The error may be found at a later point, however.

The actual error is found in the declaration of the structure `something`. A good compiler or lint will catch this error, but the underlying, though less obvious, error is flagged in the line that follows. When the compiler notes that `i` is to be assigned the value of `something.times`, the compiler is forced to take the address of `struct something` to perform the assignment. Because `something` is stored in a register, the compiler interprets this line to be in error.

You may need several minutes to find the fault. The fault is not in the assignment; the fault is in the definition.

Best register variables

What variables make the best use of the `register` class? The answer is variables that are compiler- and CPU-specific. Certain CPUs have instruction sets that make the use of `register` variables more efficient in code size and faster in program execution. `register` variables are desirable on these CPUs. Some compilers generate program code that use every spare CPU register available. These compilers can make the best choice for the use of the CPU register, and few `register` variables should be used.

The type of data that should be made a `register` variable is almost identical for all machines. As most registers in 8-, 16-, and 16-/32-bit CPUs are 16 bits wide, `register` variables for these machines should be no more than 16 bits wide. These machines can only use `char`, `short int`, `unsigned int`, `int`, and pointers. The simple data types of `float`, `double`, and `long int` generally cannot be stored in a register.

The same is true for 32-bit machines like DEC's VAX series. The 32-bit registers will not hold `float`, `double`, and `long int`.

The rationale for using register variables is to gain speed, not to improve efficiency of code size. Therefore, the best `register` variables are heavily used `int`s and pointers. Usually, `register` pointers yield better results (faster-running programs) than do `register int`s. However, the specific data types that make the best `register` variables depend on your compiler and CPU.

Experimenting is the only way to find which variables yield the best results. Try defining variables in functions with and without the `register` declaration. Time the program and compare program sizes to determine which "mix" of `register` variables is right for your CPU and compiler.

testr.c program

`testr.c` is a simple program to test the speed of programs using `register` variables. The program accepts an optional hexadecimal parameter on the basis of the following values:

```
 1 integer test
 2 short integer test
 4 pointer test
 8 parameter test
10 test register
20 test nonregister
```

For example, to run all register tests, use an argument of `1f`. To run all nonregister tests, use an argument of `2f`. To run the register integer and pointer tests, use 15.

Try running all register tests first, record the running times, and then run the nonregister tests. If both runs have the same time (within .5 seconds), your compiler does not use `register` variables. A time difference greater than .5 seconds probably indicates that your compiler does use `register` variables.

If your compiler uses `register` variables, you might again run the individual tests to catch the time difference between register and nonregister use of data types. Such tests assist in determining whether `register` variables are worthwhile.

tstr.c

```
/*      program to test register variables      */
/*                                              */
/*      use command line argument to set        */
/*      a particular test using values in       */
/*      #defines below                          */


#include <stdio.h>

#define INT             0x01
#define SINT            0x02
#define POINTER         0x04
#define PARAMETER       0x08
#define REGISTER        0x10
#define NOTREG          0x20

main(argc, argv)
int argc;
char **argv;
{

        int i, type = 0, reg = 0, noreg = 0;

        sscanf(argv[1], "%x", &type);           /* get argument */

        if(type == 0 || type == 0x10)           /* 0 or 10h */
                type = 0x1f;                    /* all register tests */
        if(type == 0x20)                        /* only 20h */
                type = 0x2f;                    /* all nonregister tests */
        if(type < 0x10)                         /* only type test specified */
                type += 0x10;                   /* make it register test */

        if(type & REGISTER)
                reg = 1;                        /* set register test flag */
        if(type & NOTREG)
                noreg = 1;                      /* set nonreg test flag */

        if(type & SINT) {       /* do short int test */
                if(reg) {
                        printf("testing register short int\n");
                        testsint(1);
                }
                if(noreg) {
                        printf("testing nonregister short int\n");
                        testsint(0);
                }
        }

        if(type & INT) {        /* do int test */
                if(reg) {
                        printf("testing register int\n");
                        testint(1);
                }
                if(noreg) {
                        printf("testing nonregister int\n");
                        testint(0);
                }
        }
```

```
        if(type & POINTER) {    /* do pointer tests (char & int) */
                if(reg) {
                        printf("testing register pointer\n");
                        testpoint(1);
                }
                if(noreg) {
                        printf("testing nonregister pointer\n");
                        testpoint(0);
                }
        }

        if(type & PARAMETER) {  /* do parameter tests */
                if(reg) {
                        printf("testing register parameters\n");
                        for(i = 0; i < 30000; ++i) {
                                testirpar(5);
                                testprpar("abcdefghij");
                        }
                }
                if(noreg) {
                        printf("testing nonregister parameters\n");
                        for(i = 0; i < 30000; ++i) {
                                testipar(5);
                                testppar("abcdefghij");
                        }
                }
                printf("parameter test complete\n");
        }

        printf("testing complete.\n");

}

int testint(mode)        /* test int */
int mode;
{

        int i;
        register int ir;
        int j;

        for(j = 0; j < 20; ++j) {
                if(mode) {
                        for(ir = 0; ir < 30000; ++ir);
                        for(ir = 30000; ir > 0; --ir);
                } else {
                        for(i = 0; i < 30000; ++i);
                        for(i = 30000; i > 0; --i);
                }
        }
        printf("integer test complete\n");

}

int testsint(mode)      /* test short int */
int mode;
{

        short int i;
        register short int ir;
        int j, k;
```

```
        for(j = 0;  j < 20;  ++j) {
                if(mode) {
                        for(k = 0;  k < 30000;  ++k)
                                i++;
                } else {
                        for(k = 0;  k < 30000;  ++k)
                                i++;
                }
        }
        printf("short integer test complete\n");

}

#define MAXARRAY 256

int testpoint(mode)             /* test pointers */
int mode;
{

        char cbuf[MAXARRAY];            /* arrays for pointer tests */
        int ibuf[MAXARRAY];
        int i;

        for(i = 0;  i < MAXARRAY - 1;  ++i) {     /* fill arrays */
                cbuf[i] = i % 256;
                if(!(i % 256))
                        cbuf[i] = 1;            /* make sure is non-null */
                ibuf[i] = i;
                if(!i)
                        ibuf[i] = 1;            /* make ibuf[0] nonzero */
        }
        cbuf[MAXARRAY - 1] = '\0';              /* null end */
        ibuf[MAXARRAY - 1] = 0;                 /* zero last element */

        testcp(cbuf, mode);
        testip(ibuf, mode);
        printf("pointer test complete\n");

}

int testcp(s, mode)             /* test character pointer */
char *s;
int mode;
{

        char *cp;
        register char *rcp;
        int i, j;

        if(mode) {
                for(i = 0;  i < 1000;  ++i) {
                        rcp = s;
                        while(*rcp) {
                                if(*rcp == 0x20);
                                ++rcp;
                        }
                }
        } else {
                for(i = 0;  i < 1000;  ++i) {
                        cp = s;
```

```
                        while(*cp) {
                                if(*cp == 0x20);
                                ++cp;
                        }
                }
        }

}

int testip(s, mode)     /* test integer pointer */
int *s;
int mode;
{

        int *ip;
        register int *rip;
        int i, j;

        if(mode) {
                for(i = 0; i < 1000; ++i) {
                        rip = s;
                        while(*rip) {
                                if(!(*rip % 256));
                                ++rip;
                        }
                }
        } else {
                for(i = 0; i < 1000; ++i) {
                        ip = s;
                        while(*ip) {
                                if(!(*ip % 256));
                                ++ip;
                        }
                }
        }

}

int testipar(n)         /* test passing integer parameter */
int n;
{

      switch(n) {
              case 0:
                      break;
              case 1:
                      break;
              case 2:
                      break;
              case 3:
                      break;
              case 4:
                      break;
              case 5:
                      break;
              case 6:
                      break;
              case 7:
                      break;
              case 8:
                      break;
```

```
                case 9:
                        break;
                case 10:
                        break;
                default:
                        break;
        }

}

int testirpar(n)        /* test passing register int parameter */
register int n;
{

        switch(n) {
                case 0:
                        break;
                case 1:
                        break;
                case 2:
                        break;
                case 3:
                        break;
                case 4:
                        break;
                case 5:
                        break;
                case 6:
                        break;
                case 7:
                        break;
                case 8:
                        break;
                case 9:
                        break;
                case 10:
                        break;
                default:
                        break;
        }

}


int testppar(s)         /* test passing character pointer parameter */
char *s;
{

        int i;

        while(*s) {
                if(*s == 'd');
                ++s;
        }

}

int testprpar(s)        /* test passing register char pointer parameter */
register char *s;
{
```

(Continued on page 16)

Quiz

Each month /c will be offering a quiz. The first correct response to arrive at our offices at /c will receive a small prize. In case of a tie, either duplicate prizes will be awarded or the earliest postmark will win. (This decision will be made at the whim of the editor!) All entries must be mailed.

Our congratulations to Mike Simonelli of Levittown, New York. Mike correctly answered our November, 1984, quiz on the ASCII characters which had no predefined meaning to C. Mike will receive a copy of *Common C Functions*.

Starting with this issue, the quiz answer will appear two issues later. This arrangement allows us to announce the winner to the quiz with the answer to the quiz.

January's quiz

Are the following two C statements identical on all 8-, 16-, and 32-bit computers?

```
x = 32768;
x = 0x8000
```

State your reason(s) for your answer

The winner of this month's quiz will receive a copy of Que's upcoming book by Kim Brand, *Common C Functions* (publication date, February, 1985).

Send your answer to the following address:

/c Quiz
c/o Que Corporation
7999 Knue Road
Indianapolis, IN 46250

Register variables

(Continued from page 15)

```
        int i;

        while(*s) {
                if(*s == 'd');
                ++s;
        }

}
```

News

(Continued from page 1)

The optional cross-compilers support the Intel 8080 and 8086 CPUs and the Motorola 68000 CPU. Each cross-compiler generates freestanding programs. The 8080 cross-compiler supports Digital Research's CP/M-80, the 8086 cross-compiler supports IBM's PC DOS, and the 68000 cross-compiler supports Digital Research's CP/M-68K.

The price of C/370 is $5,000 and includes the VM/CMS option as standard. Each cross-compiler is priced at $5,000.

Whitesmiths, Ltd.
97 Lowell Road
Concord, MA 01742
(617) 369-8499

Libraries

GKS/C is a C implementation of the Graphics Kernel System that runs on DEC VAX-11 systems or systems using the Motorola MC68000 CPU running UNIX.

Written in C, the software provides graphics capabilities from simple passive output to interactive applications that will control a variety of graphics devices. GKS/C makes use of a variety of graphics primitives, some of which allow users to address raster tube effects and to control access to special workstation characteristics. Groups of primitives can be dynamically scaled, rotated, flashed, and made visible or invisible.

Pricing starts at $2,000 for a single CPU, and $10,000 (plus $300 per CPU) for multiple-CPU licenses.

Prior Data Sciences
Suite 232
39 Highway 7
Ottawa, Ontario K2H 8R2
Canada

Documentation

AT&T Information Systems has announced a definition document for its UNIX System V. The document, available this month, is intended for those software writers who wish to port their applications across various System V implementations.

The document, *System V Interface Definition*, defines a minimum set of system calls and library functions that should be common to all operating system implementations based on UNIX System V.

A service will also be available during 1985 to help system developers conform to the System V Interface specifications. The service will test the implementation of UNIX System V for conformance to the definition.

Pricing on the document and the service will be announced later by AT&T IS.

AT&T Information Systems
Headquarters Plaza, East
1 Speedwell Avenue
Morristown, NJ 07960